Guide to Performance and Tuning: Query Performance and Sampled Selectivity

A feature of Oracle Rdb

By Claude Proteau
Oracle Rdb Relational Technology Group
Oracle Corporation

1 Improving Query Performance Using Sampled Selectivity

Oracle Rdb version 7.1.2 introduces a new optimizer feature called ***sampled selectivity***. Use of sampled selectivity can improve the performance of certain database queries by giving the Rdb optimizer more accurate information about the distribution of data in tables. The role of the Rdb optimizer is to decide how to execute a query in the most efficient manner attempting to minimize the I/O. To accomplish that, the optimizer considers different ways of joining results from various tables and different methods of retrieving the data for each table. The result of the optimization process is a ***query strategy***. The choice of query strategy depends primarily on its cost, based on estimating the number of disk I/O operations that will be done when the query is executed. This might be mitigated by constraints placed on the query, such as, fast first execution and query outlines.

2 Selectivity in the Optimization Process

Most of the time the Rdb optimizer does an excellent job of choosing query strategies. However, query optimization is not an exact science; and in a small number of cases the chosen strategies are sub-optimal (the queries take longer to perform than one might expect). Sometimes the reason for this can be traced to dramatically inaccurate estimates of the number of data rows that will be processed. Sampled rather than static computation of selectivity may help correct this.

Part of computing the I/O cost involves breaking down a SQL query's WHERE clause into individual predicates. For instance, A.LAST_NAME = B.LAST_NAME is a predicate that specifies a join condition on two tables. Another example is the predicate EMPLOYEE_ID = '00164', where column values are restricted to a range, or in this case, one particular value. The purpose of such predicates is to limit the number of rows to be retrieved. Each predicate specifies some condition that allows only those rows that qualify to be selected. The ratio of the number of qualifying rows divided by the total number of rows is called the ***selectivity factor***.

The Rdb optimizer determines predicate selectivity in different ways. For example, given a join condition, such as, A.LAST_NAME = B.LAST_NAME, the selectivity is computed as some function of the cardinality of the two tables. For predicates of the form EMPLOYEE_ID = '00164' (more particularly "column-name relational-operator literal-value" or "column-name IS NULL"), the optimizer assigns a fixed value to the selectivity, as described in the next section. It is for this second type of expression, when an index exists on the column, that sampled selectivity can be employed.

2.1 Fixed Selectivity Is Used by Default

For predicates of the form "column-name relational-operator literal-value" or "column-name IS NULL", the optimizer assigns a selectivity factor whose value depends on the operator used in the expression. In the example EMPLOYEE_ID = '00164', the default behavior of Rdb is to assign the equals operator a fixed selectivity factor of 3.125%. If the EMPLOYEES table has 100 rows in it, the equals operation is assumed on average to return three rows (which is 3%). This selectivity value is fixed and does not depend on the value '00164' in the predicate. If instead the predicate were EMPLOYEE_ID = '00273', the optimizer would still predict that three rows in the EMPLOYEES table would match that selection criterion.

Rdb provides two choices for fixed selectivity values; the first is the default set for Rdb and the other choice uses ***aggressive selectivity*** values that predict fewer rows will be returned than do the standard Rdb values. Given an estimated cost for retrieving all the data in a selected table column, selectivity is used to reduce that cost by some fraction to represent the subset of data that is of interest.

2.2 Fixed Selectivity Can Sometimes Be a Poor Predictor

Selectivity is used to predict cardinality (the number of rows to be processed), and predicted cardinality is used to estimate I/O cost. However, fixed selectivity can be a poor predictor given certain distributions of data. For example, if a table of 100 rows has a COUNTRY column and all values in the table for that column happen to be 'SWITZERLAND', the true selectivity of the predicate WHERE COUNTRY = 'SWITZERLAND' should be 100%, not 3%. This is a case where the distribution of column values over the range of possible values is very narrow. This is an example of a highly skewed data distribution.

2.3 Introducing Sampled Selectivity

Rdb introduces another way to calculate selectivity. This can be done by sampling the data in a table's index and estimating the cardinality from the actual distribution of data. Given an estimate of a predicate's cardinality and given the number of rows in a table, one can estimate the selectivity factor. Tests have shown that, on average, selectivity estimation done by sampling data in an index is more accurate than by simply using a fixed value.

Execution of the Rdb optimizer is divided into a static optimization phase and a dynamic optimization[1] phase. The role of the static optimizer is to choose the "best" query strategy. For certain types of queries the dynamic optimizer processes several competing indices. When multiple indexes on a table exist, the dynamic optimizer accesses all of them to see which is the most productive. Different executions of the query can thus adapt to changing input parameters and use the best index.

The first step in dynamic optimization involves ordering background indexes by the number of expected returned keys. The number of index entries to be processed is estimated by sampling each candidate index. The static optimizer now uses that same method of sampling an index to estimate the cardinality of a result and from that to compute the selectivity.

2.4 Pros and Cons of the Sampled and Fixed Selectivity Methods

Sampled selectivity computation is usually more accurate than using a fixed selectivity value because it is based on the actual distribution of data in a table. A sorted, ranked index gives better overall results than does a sorted, non-ranked index because cardinality information is stored within the index. For any given data value, anyone of the three methods (fixed, sorted, ranked) might yield the most accurate result. Typically, ranked indexes give the best results and fixed selectivity gives the least accurate results.

The following is a simple range query on the EMPLOYEES table in the sample PERSONNEL database.

```
SQL> select employee_id from employees where employee_id > ‘nnnnn’;
```

There are 100 rows in the EMPLOYEES table. The values in the EMPLOYEE_ID column are unique and range from ’00164’ to ’00471’. The worst estimates for all three methods occur when ‘nnnnn’ = ‘00164’. The fixed selectivity method predicts 35 rows, so it is too low by 65 rows. The sorted, unranked index method predicts 34 rows, so it is too low by 66 rows. The ranked index method predicts 84 rows, but it is only wrong by 16 rows.

Sampled selectivity comes with a small cost. In order to get the more accurate estimates, the optimizer must sample indexes during query compilation; and this might require I/O operations to be performed depending on how much of the index nodes are already in memory. If the indexes

[1] Dynamic optimization was first introduced in Rdb V4.0.

used for the estimation are also used during query execution, there might be no additional I/O if the same index nodes must be referenced again as the index information will typically remain buffered. Also, if the query is compiled once but executed many times, any additional I/O to perform the estimation might be insignificant.

There is no evidence to show that enabling aggressive selectivity for all queries will result in overall better query strategies than by using the standard (default), fixed Rdb values. Aggressive selectivity is best used for specific queries where it is shown to help and where sampled selectivity cannot be used.

2.5 Requirements for Using Sampled Selectivity

When the feature is enabled, sampled selectivity estimation is attempted only for certain forms of predicate and only under the right set of circumstances. For example, if a table has no indexes, selectivity cannot be estimated by sampling since there are no indexes on which to sample the data. The following is a set of rules that define when sampled selectivity estimation can and cannot be performed.

- **Predicate Form**

  The predicate must be one of the following types:

  column = literal
  column <> literal
  column > literal
  column >= literal
  column < literal
  column <= literal
  column IS NULL

  For all but the IS NULL case, the operands can be transposed, for example, literal = column. When predicates use variables instead of literals, estimation by sampling cannot be done because the value is unknown at query compilation time.

- **Base Table Column**

  The column reference must be to a base table column, such as the EMPLOYEE_ID column in the EMPLOYEES table, or to a view column that maps one-to-one with such a column.

- **Column Is First Index Segment**

  At least one sorted or sorted, ranked index on the table must exist with the predicate's column as its first segment. The index may have more than one segment.

- **Hashed Index not Used**

  For an index to be useful in the estimation process, it must be either a sorted index or a sorted, ranked index. Hashed indexes are not used for sampled selectivity estimation.

- **Single Partition Index**

  For an index to be considered useful, it can only have a single partition.

- **Ascending Key Values**

  In the candidate indexes, all columns must be sorted in ascending order.

- **No Explicit Collating Sequence**

  The column used for the first index segment must not have any explicit collating sequence specified.

- **No Mapping Values**

  The first index segment must not be mapped (see the MAPPING VALUES clause in the CREATE INDEX statement).

In the future it might be possible to relax or eliminate some of the preceding restrictions given sufficient interest in so doing.

3 How to Enable the Various Selectivity Methods

Rdb uses several different methods for estimating predicate selectivity. These methods are shown in Table 2 and the programmer can influence the optimizer to select two methods: (Fixed) and

(Index). By default Rdb will use its standard set of values for assigning (Fixed) selectivity to predicates. There are several ways to specify the method to be used:

- For individual queries:

  The type of selectivity computation to use for individual queries can be specified by including the optional OPTIMIZE WITH clause on INSERT ... SELECT, SELECT, UPDATE, DELETE, and compound statements (only on the outermost BEGIN-END block). See Section 3.1.

- For queries made within the current SQL session:

  Establish a default method for selectivity calculation to avoid having to include the OPTIMIZE WITH clause on each SQL query: for interactive and dynamic SQL use the SET OPTIMIZATION LEVEL statement (see section 3.2) or the SET FLAGS statement (see section 3.3). For pre-compiled SQL and for SQL module language code use the OPTIMIZATION_LEVEL qualifier, see section 3.4.

- For all query sessions:

  See Section 3.3 for the use of the RDMS$SET_FLAGS logical name.

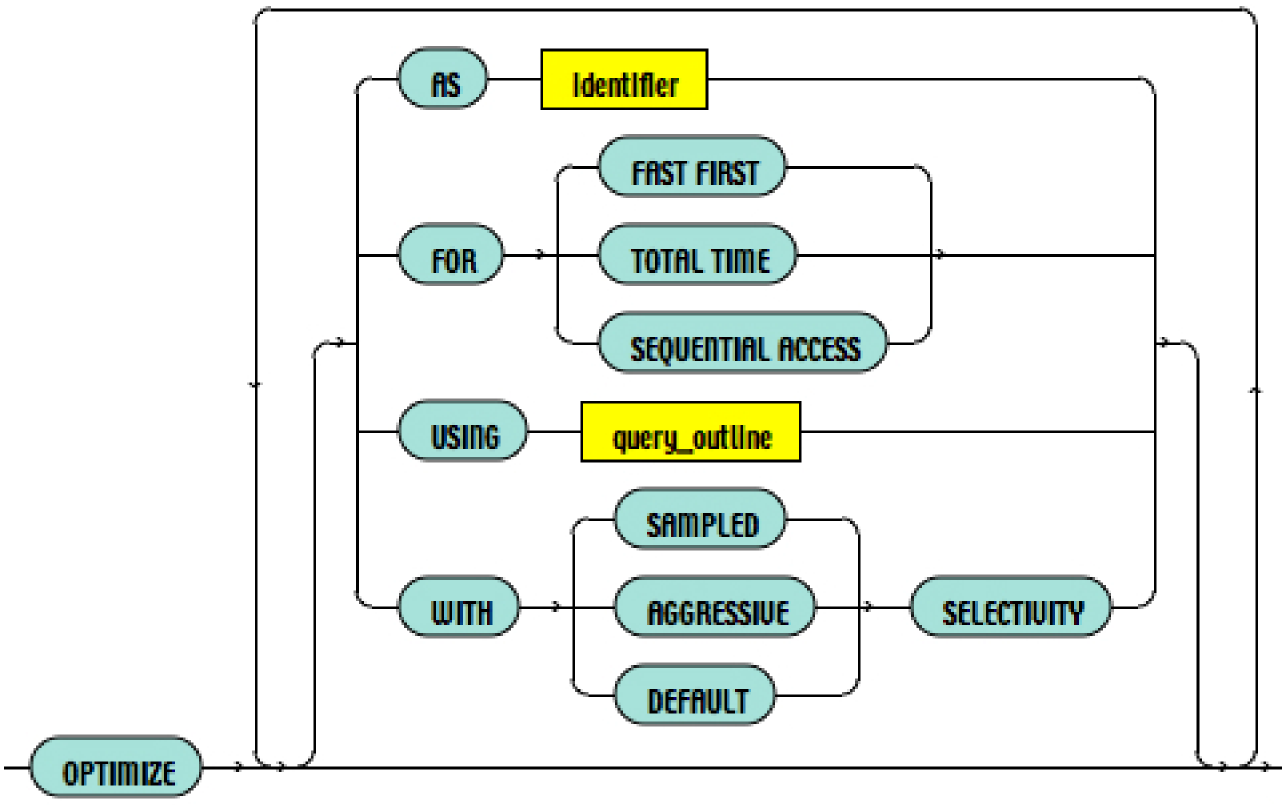
- For RMU/UNLOAD:

  A qualifier has been added to the RMU/UNLOAD command that allows the programmer to specify how selectivity is to be evaluated. See Section 3.5.

3.1 OPTIMIZE WITH Clause

The INSERT ... SELECT, SELECT, UPDATE, DELETE and compound statements have an optional clause, OPTIMIZE WITH, that specifies what type of selectivity computation method is to be used. The OPTIMIZE FOR, OPTIMIZE USING, and OPTIMIZE AS forms of the OPTIMIZE clause are already described in the Oracle Rdb SQL Reference Manual.

Syntax:



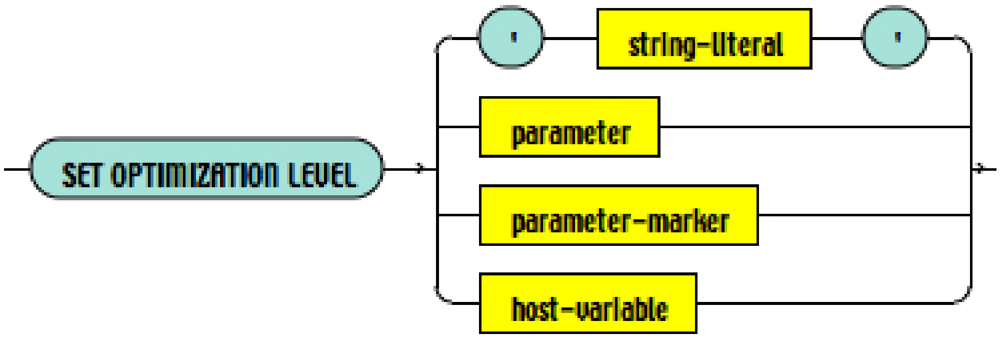
When using the OPTIMIZE WITH clause, one of three options can be specified.

- **sampled selectivity**: the Rdb optimizer will use the index sampling method for selectivity estimation wherever possible. For those predicates where this is not possible, the optimizer will revert to use standard, fixed selectivity values.
- **aggressive selectivity**: the Rdb optimizer will use the fixed, aggressive values for selectivity computation.
- **default selectivity**: this specifically means that index sampling and aggressive selectivity will not be used.

3.2 SET OPTIMIZATION LEVEL Statement

New options have been added to the SET OPTIMIZATION LEVEL statement, options that allow the programmer to specify default selectivity behavior for INSERT ... SELECT, SELECT, UPDATE, DELETE and compound statements within an interactive or dynamic SQL session.

Syntax:



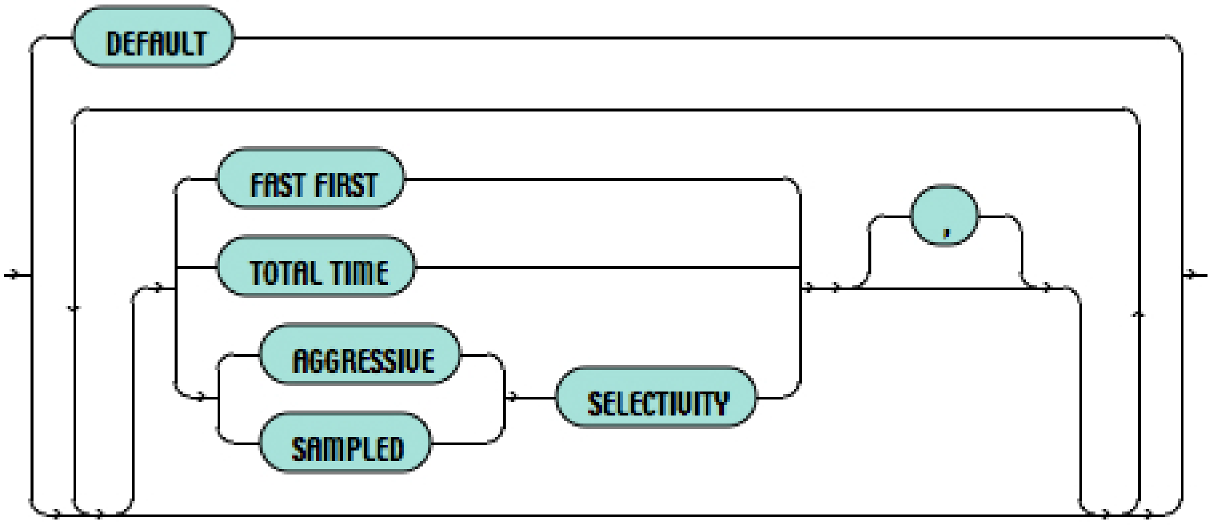
Options are provided for the SET OPTIMIZATION LEVEL command as a string literal, or at run-time they can be provided as parameters. If more than one option is required the keywords passed to the SET OPTIMIZATION LEVEL statement must be separated by a comma. Only one of FAST FIRST[2] and TOTAL TIME, and only one of AGGRESSIVE SELECTIVITY and SAMPLED SELECTIVITY may be specified.

For selectivity computation the chosen default is applied to each INSERT ... SELECT, SELECT, UPDATE, DELETE, and compound statement in the SQL session provided that those statements do not explicitly have the OPTIMIZE WITH clause, described in Section 3.1.

If sampled selectivity is specified, the Rdb optimizer will use the index sampling method for selectivity estimation wherever possible. For those predicates where this is not possible, Rdb will revert to using standard, fixed selectivity values. If aggressive selectivity is selected, the Rdb optimizer will use the fixed, aggressive values for selectivity computation.

[2] The FAST FIRST, TOTAL TIME and DEFAULT options of the SET OPTIMIZATION LEVEL statement are described in the Oracle Rdb SQL Reference Manual.

Syntax:



3.3 The SELECTIVITY Debug Flag

Yet another way to declare how selectivity is to be computed is by setting the SELECTIVITY debug flag. This can be done in one of two ways: defining the OpenVMS logical name, RDMS$SET_FLAGS, or using the SET FLAGS statement[3].

When selectivity is defined using the RDMS$SET_FLAGS logical name, it affects queries for all SQL sessions which are run within the scope of that logical name. Doing so can be useful when trying to debug query performance problems. When selectivity is enabled using the SET FLAGS statement, its effect lasts for the duration of the database attach.

The SELECTIVITY debug flag can be used to specify default, aggressive, or sampled selectivity behavior. In addition, the SELECTIVITY flag allows both aggressive and sampled behavior to be enabled, something that is not possible with the SET OPTIMIZATION LEVEL statement or the OPTIMIZE WITH clause. The SELECTIVITY debug flag affects queries that do not otherwise have a selectivity mode specified. It can also affect partial query outlines, triggers, constraints, and internal Rdb queries.

[3] The RDMS$SET_FLAGS logical name and the SET FLAGS statement are described in the Oracle Rdb SQL Reference Manual.

The SELECTIVITY debug flag takes a numeric argument, with a value from 0 to 3. For example,

```
$ DEFINE RDMS$SET_FLAGS "SELECTIVITY(2)"
```

or

```
SQL> SET FLAGS 'SELECTIVITY (2)';
```

Table 1 shows the numeric values for the SELECTIVITY debug flag and their meanings.

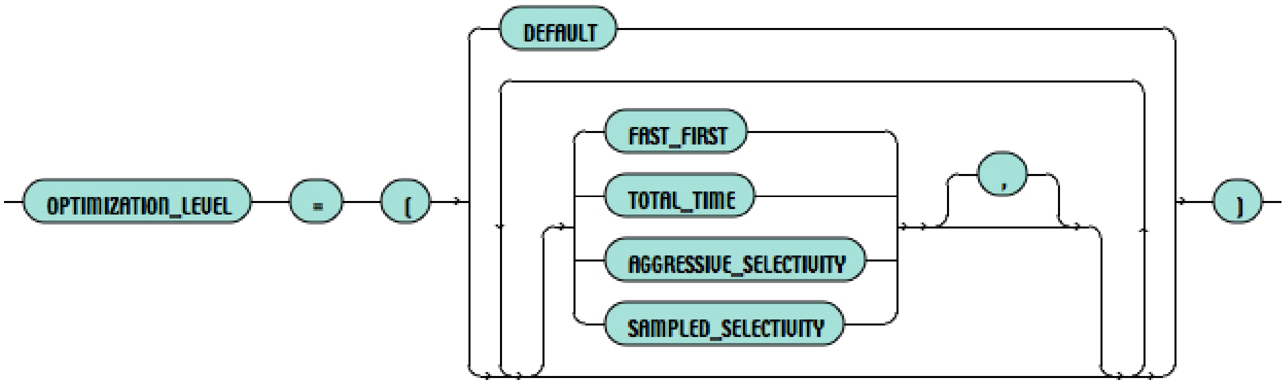
Table 1: Settings for the SELECTIVITY Debug Flag

Selectivity Debug Flag Setting	Meaning
SELECTIVITY (0)	Default selectivity
SELECTIVITY (1)	Aggressive selectivity
SELECTIVITY (2)	Sampled selectivity
SELECTIVITY (3)	Sampled + aggressive selectivity

3.4 SQL Precompiled and SQL Module Language Code

Optimizer selectivity controls can also be enabled using the OPTIMIZATION_LEVEL qualifier on SQL precompiled or SQL Module Language compiled code. In addition to being able to establish default values for TOTAL TIME versus FAST FIRST optimization, the OPTIMIZATION_LEVEL qualifier can now indicate the type of selectivity estimation to perform by default. SELECT, INSERT ... SELECT, UPDATE, DELETE and compound statements (on the outermost BEGIN ... END) will inherit these settings during compilation of the module.

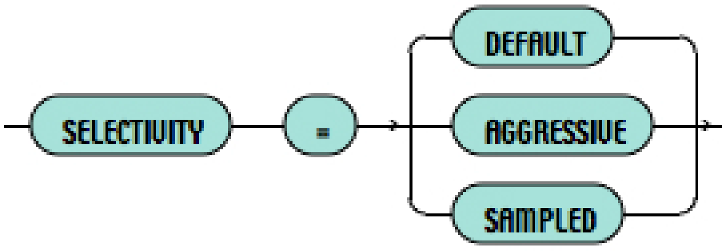
Syntax:



3.5 RMU/UNLOAD/OPTIMIZE

A SELECTIVITY option has also been added to the RMU/UNLOAD/OPTIMIZE qualifier. By specifying the SELECTIVITY option the programmer can direct how the optimizer will estimate predicate selectivity values during RMU/UNLOAD operations.

Syntax:



If sampled selectivity is chosen, the Rdb optimizer will use the index sampling method for selectivity estimation wherever possible. For those predicates where this is not possible, the optimizer will revert to use standard, fixed selectivity values. If aggressive selectivity is chosen, the Rdb optimizer will use the fixed, aggressive values for selectivity computation. Finally, if default selectivity is chosen, this specifically means that index sampling and aggressive selectivity will not be used.

```
$ RMU/UNLOAD/OPTIMIZE=(SELECTIVITY=SAMPLED) mydatabase mytable mydata
```

3.6 Improving the Accuracy of Sampled Selectivity

Sampled selectivity estimation using a sorted, ranked index can be done with varying degrees of accuracy. More accuracy might require that more I/O be spent during the index sampling process, depending on how many of the index nodes are resident in memory. The REFINE_ESTIMATES debug flag controls sampling accuracy. For more information about refining estimates on sorted, ranked indexes, see *Guide to Database Performance and Tuning: Predicate Estimation*[4].

When sampled selectivity is being computed with refined estimates in effect, there is no specific numeric limit placed on the number of I/O operations to be performed. That being the case, there are three refinement rules (as defined in the *Predicate Estimation* article) that can affect the calculation of sampled selectivity:

- Rule 3 – Limit refinement to the TRUE/MIXED cardinality case.

[4] Available from the Oracle Rdb Technical Journal.

To enable this refinement rule, SET FLAGS 'REFINE_ESTIMATES(4)' or DEFINE RDMS$SET_FLAGS "REFINE_ESTIMATES(4)".

- Rule 4 – Limit refinement so that the error in the estimate is within ten percent.

  To enable this refinement rule, SET FLAGS 'REFINE_ESTIMATES(8)' or DEFINE RDMS$SET_FLAGS "REFINE_ESTIMATES(8)".

- Rule 5 – Try to provide precise estimates.

  To enable this refinement rule, SET FLAGS 'REFINE_ESTIMATES(16)' or DEFINE RDMS$SET_FLAGS "REFINE_ESTIMATES(16)".

Setting the value of the REFINE_ESTIMATES flag to 12 combines rules 3 and 4. Rule 5 is only needed if no other limits on the refinement process are required and the most precise estimates are desired.

4 Details about the Sampled Selectivity Process

This section gives details about the operation of the Rdb optimizer as it performs sampled selectivity computation.

In an early stage of query compilation the static optimizer scans each WHERE clause in the query, locates each ***leaf predicate***, and assigns it a selectivity factor. Consider the following: WHERE LAST_NAME = 'Toliver' AND FIRST_NAME = 'Alvin'. Also, assume there exist sorted indexes on the LAST_NAME and FIRST_NAME columns. The leaf predicates are (1) LAST_NAME = 'Toliver', and (2) FIRST_NAME = 'Alvin'. There is a higher-level predicate, the AND of two expressions, which is not a leaf predicate. In this example, sampled selectivity estimation is only performed for the two leaf predicates. The key steps in this process are:

1. Validate the predicate format

   The form of the predicate must be one of those described in Section 2.5.

2. Avoid Redundant Estimation

   To avoid unnecessary I/O, the optimizer maintains a list of up to 100 predicates for which estimation by sampling has already been done. The list only applies to the current query. If two predicates are the same and are for the same table, the second predicate is given the already-computed selectivity value of the first one.

3. Verify the Presence of One or More Useful Indexes

   The table in which the predicate column exists must have one or more indexes which could be used for sampled selectivity estimation. What determines that an index is useful is explained in Section 2.5.

4. Choose the Best Index for the Job

   A table can have multiple indexes, and several indexes might be valid for doing the estimation. The optimizer looks at each such index and chooses one using the following criteria. (1) Ranked over Non-Ranked: First, a sorted, ranked index is assumed to give more accurate estimates than an index that is not. (2) Unique over Duplicates Allowed: If both indexes are equal thus far, and if one index is unique and the other index allows duplicates, the unique index is chosen as likely to give the more accurate results. (3) Shorter index key Length: All other things being equal, an index with a shorter index key length is chosen.

5. Estimate Cardinality by Sampling the Index

   Once the optimizer chooses an index it scans the index to estimate the expected cardinality for the predicate. Cardinality is then used to compute selectivity.

6. When Sampled Estimation Fails

   If sampled selectivity estimation is attempted and fails for whatever reason, the optimizer reverts to using the fixed selectivity values.

5 Diagnostic Information about Selectivity Estimation

The Rdb optimizer can display, for each query that is compiled, the selectivity values that were used, the methods which were chosen to derive those values, and cardinality predicted for each predicate (where appropriate), the name of the index used to estimate selectivity, or the reason(s) that selectivity could not be estimated by sampling.

5.1 How to Enable Diagnostic Output

Normally, diagnostic information is not shown. To see information about the optimizer's query estimation process, the ESTIMATES flag must be enabled. To view details about predicate selectivity computation, the DETAIL flag must also be enabled. These flags are set using either the RDMS$SET_FLAGS logical name or the SET FLAGS statement in SQL.

Normal query estimation summary:

To enable standard output about the query estimation process, define the RDMS$SET_FLAGS logical name as follows:

```
$ DEFINE RDMS$SET_FLAGS ESTIMATES
```

or

```
SQL> SET FLAGS 'ESTIMATES';
```

Examples of the output for this and other flag settings are shown in Section 5.4.

Query estimation summary plus predicate selectivity:

To enable detailed output about the query estimation process, define the RDMS$SET_FLAGS logical name as follows:

```
$ DEFINE RDMS$SET_FLAGS "ESTIMATES,DETAIL(2)"
```

or

```
SQL> SET FLAGS 'ESTIMATES,DETAIL(2)';
```

With these settings the standard summary about query estimation plus, for each predicate, the selectivity values used, what methods were chosen to derive those values, and cardinality predicted (where appropriate).

Query estimation summary plus predicate selectivity and more:

If DETAIL(3) is used in the preceding statements, then either the name of the index used to estimate selectivity, or the reason(s) that selectivity could not be estimated by sampling will be displayed.

5.2 How to Disable Diagnostic Output

To disable output about the query estimation process, either use the RDMS$SET_FLAGS logical name or use the SET FLAGS statement in SQL.

De-assigning the RDMS$SET_FLAGS logical will ensure that no further sessions use those flag settings. Note that logical names can be defined in various logical name tables.

```
$ DEASSIGN RDMS$SET_FLAGS
```

If sampled selectivity is still required but query estimates are no longer needed then redefine the logical with the required options:

```
$ DEFINE RDMS$SET_FLAGS “SELECTIVITY(2)”
```

When redefining the RDMS$SET_FLAGS logical name, include those flags still remaining in effect. By contrast, in interactive or dynamic SQL the negated keyword can disable those flags.

```
SQL> SET FLAGS ‘NOESTIMATES,NODETAIL’;
```

The preceding statement disables query estimates but leaves other flags unchanged.

5.3 Details about Selectivity Estimation Diagnostics

The method for selectivity computation appears in parentheses on the output line showing estimated selectivity (see examples in Section 5.4). Table 2 shows the various methods:

Table 2: Methods of Computing Selectivity

Method	Meaning
(Average)	Selectivity was computed using some average statistical value known about the table or an index, e.g., an index segment group factor.
(Cardinality)	Selectivity was computed as some function of current table cardinality.
(Fixed)	A fixed value for selectivity was chosen (either standard or aggressive).
(Index)	Selectivity was computed by sampling an index.
(Index) (Dup)	Selectivity computation was avoided. The predicate is a duplicate of one elsewhere in the query, one for which selectivity was computed by sampling an index.

When selectivity cannot be determined by index sampling, and when the level of detail in the diagnostic output is properly set (see Section 5.1), messages can appear in the output to explain the reason(s). These messages and what they mean are listed in Table 3.

Table 3: Reasons That Selectivity Sampling Cannot Be Done

Message	Meaning
Column has an explicit collating sequence	This is explained in Section 2.5.
Sampled selectivity is disabled	This is self-explanatory.
Error during index scan	It was not possible to perform the index scan. For example, under unusual circumstances a buffer overflow can occur.
Exception error	An unexpected error occurred. This indicates that a design or implementation problem exists in the Rdb optimizer. The problem should be reported to Oracle.
Expression not supported for sampled selectivity	See Section 2.5 to see which expressions are acceptable and which are not.
Indexes not valid for sampled	For selectivity to be computed by sampling, the

selectivity	predicate column's table must have an index with appropriate attributes, as explained in Section 2.5.
Internal error …	An unexpected error occurred. This indicates that a design or implementation problem exists in the Rdb optimizer. The problem should be reported to Oracle.
Operator not supported for sampled selectivity	See Section 2.5 to see which operators are acceptable and which are not.
Selectivity is fixed for outer join Booleans	The selectivity of an outer join Boolean predicate is fixed at 1.0 (i.e., 100%).
Table has no indexes	For selectivity to be computed by sampling, the predicate column's table must have an index with appropriate attributes, as explained in Section 2.5.

5.4 Examples of Selectivity Estimation Diagnostics

This section shows examples of what might be observed when query estimation diagnostics are requested.

Example 1: Normal diagnostics for the query estimation process

The following example shows a simple query and the normal summary one can see for the query estimation process. This occurs when the level of detail is not specified (equivalent to detail level 0). The output of the estimates summary is described in the Oracle Rdb7 Guide to Database Performance and Tuning, Section C.4, Displaying Optimization Statistics with the O Flag.

```
SQL> set flags ‘estimates’;
SQL> select last_name from employees where employee_id > '00400';
…
Solutions tried 2
Solutions blocks created 1
Created solutions pruned 0
Cost of the chosen solution   3.3090658E+00
Cardinality of chosen solution   3.5000000E+01
```

Example 2: Detail level 2 estimates

There is no difference between detail levels 0 and 1 for the estimation diagnostics. At detail level 2, prior to the normal estimates, Rdb displays information about the predicates in the query, in this case, the predicate EMPLOYEE_ID > '00400'. First, each table is listed along with a correlation number. Below, the EMPLOYEES table is given as table 0. Next, each predicate is shown (this query has only one). 0.EMPLOYEE_ID signifies the EMPLOYEE_ID column in table 0, i.e., EMPLOYEES. Following the listed predicate is a line showing the predicate selectivity, the method by which that selectivity value was derived (Index, in this case), and the estimated cardinality for the predicate. Estimated cardinality and selectivity are related by the formula:

selectivity = estimated cardinality / table cardinality

The notation (Index) means that predicate selectivity was calculated by sampling an index (sampled selectivity).

```
SQL> set flags ‘estimates,detail(2)’;
SQL> select last_name from employees where employee_id > '00400'
       optimize with sampled selectivity;
…
~Predicate selectivity and cardinality estimation:
Tables:
  0 = EMPLOYEES
Predicates:
  0.EMPLOYEE_ID > '00400'
    Selectivity  5.9999999E-02 (Index)        Estimated cardinality 6
Solutions tried 2
Solutions blocks created 1
Created solutions pruned 0
Cost of the chosen solution   1.9074016E+00
Cardinality of chosen solution   6.0000000E+00
```

Example 3: Detail level 3 estimates

At detail level 3, the output includes additional information: either (1) the name of the index used to estimate selectivity, or (2) the reason(s) that selectivity could not be estimated by sampling. In this example there are two predicates. For the first predicate, selectivity is computed by index sampling and the name of the index used is shown. For the second predicate, selectivity cannot be computed by sampling because there is no useful index for doing so. Although there is an index on the EMPLOYEE_ID column, there is none on the LAST_NAME column.

```
SQL> set flags ‘estimates,detail(3)’;
SQL> select last_name from employees
cont> where employee_id < ‘00180’ and last_name > 'W'
      optimize with sampled selectivity;
```

```
~Predicate selectivity and cardinality estimation:
Tables:
  0 = EMPLOYEES
Predicates:
  0.EMPLOYEE_ID < '00180'
    Selectivity  1.6000000E-01 (Index)        Estimated cardinality 16
      Index EMP_EMPLOYEE_ID used
  0.LAST_NAME > 'W'
    Selectivity  3.4999999E-01 (Fixed)        Estimated cardinality 35
      Indexes not valid for sampled selectivity
        Index EMP_EMPLOYEE_ID is not useful
          First index segment is not predicate column
Solutions tried 2
Solutions blocks created 1
Created solutions pruned 0
Cost of the chosen solution   2.4074016E+00
Cardinality of chosen solution   5.5999999E+00
```

Example 4: Example showing the (Average) Method for Computing Selectivity

For a predicate of the form column = value, selectivity is determined by sampling if possible, that is, the (Index) method. If this cannot be done and if the table has a single segment index on that column, the method used is (Average). That is, an average value is used for the entire index.

```
SQL> set flags 'estimates,detail(3)';
SQL> select employee_id, last_name from employees
cont> where employee_id = '00250';

~Predicate selectivity and cardinality estimation:
Tables:
  0 = EMPLOYEES
Predicates:
  0.EMPLOYEE_ID = '00250'
    Selectivity  9.9999998E-03 (Average)      Estimated cardinality 1
      Sampled selectivity is disabled
Solutions tried 2
Solutions blocks created 1
Created solutions pruned 0
Cost of the chosen solution   1.6474016E+00
Cardinality of chosen solution   1.0000000E+00
```

Example 5: Example showing the (Cardinality) Method for Computing Selectivity

For some predicates, the selectivity is a function of table cardinality.

```
SQL> select e.employee_id, e.last_name, d.department_name
cont> from employees e, departments d
cont> where e.employee_id = d.manager_id;

~Predicate selectivity and cardinality estimation:
Tables:
  0 = EMPLOYEES
  1 = DEPARTMENTS
Predicates:
  0.EMPLOYEE_ID = 1.MANAGER_ID
    Selectivity  9.9999998E-03 (Cardinality)
      Expression not supported for sampled selectivity
Solutions tried 10
Solutions blocks created 4
Created solutions pruned 1
Cost of the chosen solution   4.5832439E+01
Cardinality of chosen solution   2.6000000E+01
```

Oracle Rdb
Guide to Perform and Tuning: Query Performance and Sampled Selectivity
February 2004

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Oracle Corporation provides the software
that powers the Internet.